VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

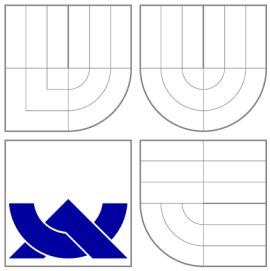
MĚŘENÍ VÝKONNOSTI BALÍČKU JAVA.MATH

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

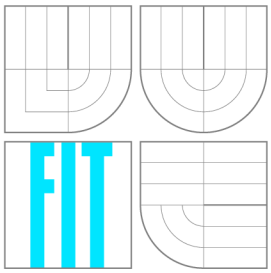
AUTOR PRÁCE
AUTHOR

PAVEL FRÝZ

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

MĚŘENÍ VÝKONNOSTI BALÍČKU JAVA.MATH

JAVA.MATH PACKAGE BENCHMARK

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PAVEL FRÝZ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. VENDULA HRUBÁ

BRNO 2014

Abstrakt

Cílem této práce je porovnat výkonnost jednotlivých implementací java virtuálních strojů při užití balíčku java.math. Jsou popsány třídy BigInteger a BigDecimal a existující nástroje pro výkonnostní testování java virtuálních strojů. V textu práce je zdokumentovaná použitá testovací sestava a v rámci práce vytvořený nástroj na měření výkonnosti implementace tříd BigInteger a BigDecimal. Jsou uvedeny a analyzovány výsledky měření tímto nástrojem.

Abstract

The aim of this work is to compare the performance of different implementation of java virtual machine, when using package java.math. The classes BigInteger and BigDecimal are described. Existing benchmarks for measuring the performance of a java virtual machine me are also described. The work also describes in detail the hardware and operation system used for performing benchmarks. A tool for the benchmarking performece of implementation classes BigInteger and BigDecimal has been developed. Finally, the analysis and the discussion upon the results benchmarks has been made.

Klíčová slova

výkonnostní testování, java, balíček java.math, BigInteger, BigDecimal

Keywords

performance testing, java, package java.math, BigInteger, BigDecimal

Citace

Pavel Frýz: Měření výkonnosti balíčku java.math, bakalářská práce, Brno, FIT VUT v Brně, 2014

Měření výkonnosti balíčku java.math

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením paní Ing. Venduly Hrubé. Další informace mi poskytl pan Ing. Pavel Tišnovský, Ph.D. ze společnosti RedHat. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Pavel Frýz
21. května 2014

Poděkování

Zde bych chtěl poděkovat paní Vendule Hrubé za odborné vedení této práce a dále panu Pavlu Tišnovskému z firmy RedHat, který mi poskytl odbornou pomoc.

Obsah

Kapitola 1

Úvod

Výsledky měření výkonnosti jsou důležitou součástí každé aplikace. Mohou poukázat na slabé stránky a vést k jejich zlepšení. Od prvního vydání z roku 1995 prošla Java obrovským pokrokem. Například Java 1.1.8 byla osmkrát rychlejší než java 1.0, což bylo do značné míry způsobeno zavedením JIT (just-in-time) kompilátoru ve verzi 1.1.8. [7].

Tato práce se zabývá právě výkonnostním testováním java virtuálních strojů. A je zaměřena na porovnání java virtuálních strojů při použití tříd z balíčku java.math, na kterých závisí výkonnost mnoha aplikací. Jejich využití můžeme najít v aplikacích, které provádějí finanční transakce, nebo také v šifrovacích algoritmech, které zabezpečují bezpečnost dat.

Cílem této práce je tedy vyvinout nástroj, který by umožňoval provést měření výkonnosti jednotlivých strojů, a na základě výsledků měření zjistit výhody a nevýhody jednotlivých implementací java virtuálních strojů, které jsou v současnosti často používány.

Přehled stávajících nástrojů na měření výkonnosti java virtuálních strojů a jejich rozdělení najdete v kapitole 2. Popis balíčku java.math a tříd BigInteger a BigDecimal, které jsou součásti tohoto balíčku, je zahrnut v kapitole 3.

V kapitole 4 je stručně popsána implementace nástroje pro měření výkonnosti implementací tříd BigInteger a BigDecimal naprogramovaného v Javě s použitím JMH knihovny. V kapitole najdete přehled potenciálních problému při výkonnostním testování a jejich řešení.

V kapitole 5 najdete analýzu výsledku testů, které jsou získány spuštěním tohoto nástroje na několika implementacích java virtuálního stroje. V kapitole najdete porovnání těchto strojů.

V závěru se diskutuje stav práce, dosažených výsledků a možnost dalšího využití a použití vytvořeného nástroje.

Kapitola 2

Nástroje na měření výkonnosti JVM

V následující kapitole najdete přehled kategorií, do kterých můžeme zařadit jednotlivé nástroje na měření výkonnosti. Dále najdete přehled stávajících nástrojů na měření výkonnosti implementací java virtuálních strojů (JVM). U každého nástroje najdete přehled výkonnostních testů, které jsou jeho součástí, a každý nástroj bude klasifikován do popsaných kategorií.

2.1 Rozdělení nástrojů na měření výkonnosti

Dle [7] můžeme nástroje pro měření výkonnosti rozdělit do tří kategorií, mikrobenchmarky, makrobenchmarky a mezobenchmarky. Kde každá skupina má svoje klady a zápory.

2.1.1 Mikrobenchmarky

Mikrobenchmarky slouží k měření výkonu velmi malé specifické části aplikace. Například může sloužit k porovnání volání synchronní a asynchronní metody nebo k porovnání různých implementací aritmetické operace.

Jejich značnou nevýhodou je to, že se stejný kód může chovat jinak při izolovaném testování, než při skutečném použití v reálné aplikaci, viz kapitola 4.

2.1.2 Makrobenchmarky

Makrobenchmarky naopak slouží k měření výkonu rozsáhlého kódu. Svým rozsahem a chováním se snaží co nejvíce přiblížit reálné aplikaci.

Výsledky měření tedy více odpovídají reálnému nasazení oproti mikrobenchmarkům, jelikož výkonnost komplexního systému, se nerovná sumě výkonností jednotlivých částí.

2.1.3 Mezobenchmarky

Poslední skupinou jsou pak mezobenchmarky. Které se podobají mikrobenchmarkům tím, že jsou zaměřeny na testování specifické části aplikace. Ale svým rozsahem se více podobají makrobenchmarkům.

Mezobenchmarky mají méně nedostatků než mikrobenchmarky a lépe se s nimi pracuje než s makrobenchmarky. Jedná se o rozumný kompromis mezi mikrobenchmarkem a makrobenchmarkem, ale nemůže nahradit měření reálné aplikace.

2.2 SPECjvm2008

SPECjvm2008 je nástroj vytvořený společností SPEC (Standard Performance Evaluation Corporation), jedná se o neziskovou organizaci, která sdružuje více jak 80 předních světových společností, vzdělávacích institucí a vládních agentur. Vývoje se účastnily společnosti AMD, BEA, HP, IBM, Intel a Sun. SPECjvm2008 je náhrada za SPECjvm98 [15].

Nástroj umožňuje porovnání výkonností různých implementací java virtuálních strojů. Skládá se z několika testů, které zatěžují různé součásti systému.

Z hlediska předchozího rozdělení bych jednotlivé testy, které jsou součásti nástroje, klasifikoval jako makrobenchmarky.

2.2.1 Jednotlivé testy

Compiler

Tento test využívá OpenJDK kompilátor pro kompilaci množiny zdrojových souborů. Jedná se o zdrojové soubory kompilátoru javac a zdrojových souborů testu sunflow, který je popsán dále. Používá vlastní správce souborů, který využívá práci s pamětí na místo přístupu k pevnému disku a souborovým operacím.

Compress

Tento test provádí kompresi dat, použitím upravené metody Lampel-Ziv. Jde o obdobu test 129.compress z nástroje SPEC CPU95, ale na místo syntetických dat používá reálná data ze souboru.

Crypto

Test je zaměřen na různé oblasti kryptografie a je rozdělen do tří menších testů.

Derby

Jedná se o náhradu db testu z nástroje SPECjvm98. Využívá open-source databázi napsanou v jazyce Java. Test je zaměřen na výpočty nad datovým typem BigDecimal a na databázovou logiku. Výpočty se snaží být mimo rozsah 64 bitů, kdy je pro vnitřní uložení čísel často použit datový typ long.

MPEGaudio

Tento test dekóduje hudební soubory, které vyhovují specifikaci ISO MPEG Layer-3. Velikost hudebních souborů je přibližně 4 MB. Test zatěžuje čísla s plovoucí řádovou čárkou.

Scimark

Jedná se o soubor testů vytvořených NIST, viz 2.3.

Serial

Tento test provádí serializaci a deserializaci primitivních datových typů a objektů. Test je typu klient-server, kde serializovaná data klienta jsou poslána na server, který je následně deserializuje. Vše probíhá v rámci jednoho systému. Zatěžuje metodu Object.equals().

Sunflow

Testuje grafickou vizualizaci, používá open source knihovnu Sunflow, vizualizace probíhá paralelně v několika vláknech a je při ní použita globální osvětlovací metoda.

XML

Test je složen ze dvou částí. V první části se testuje implementace javax.xml.transform a přidružených rozhraní. Součástí testu je formátování XML dokumentu pomocí XSL stylu. V druhé části se testuje implementace javax.xml.validation a přidružených rozhraní. Test provádí validaci XML dokumentu vůči XML schématu.

2.3 SciMark 2.0

Scimark 2.0 je nástroj vyvinutý institutem NIST (National Institute of Standards and Technology), na vývoji se podíleli pánové Roldan Pozo a Bruce Miller. Cílem bylo lepší pochopení chování JVM na různých platformách při numerických výpočtech [6].

Nástroj se skládá z pěti testů: FFT, Gauss-Seidel relaxation, Sparse matrix-multiply, Monte Carlo integration a LU factorization. Z výsledku jednotlivých testů je poté složeno celkové skóre v jednotkách Mflops (milion operací s čísly s plovoucí řádovou čárkou za sekundu).

Jednotlivé testy jsou zvoleny tak, aby co nejlépe ukázaly výkonnost JVM při vědeckých a numerických výpočtech. Každý test obsahuje dva datové vzorky, malý a velký. Malý vzorek se snaží odstranit vliv, přístupu do paměti a je zaměřen na vnitřní implementaci JVM a využití procesoru. Velký vzorek je zaměřen na výkon paměťového podsystému. Testy svým rozsahem a zaměřením odpovídají mezobenchmarkům.

2.3.1 Jednotlivé testy

Fast Fourier Transform

Test provádí rychlou Fourierovu transformaci komplexních čísel. Test je zaměřen na výpočty s komplexními čísly, práci s pamětí a trigonometrické funkce.

Malý vzorek obsahuje 4 000 komplexních čísle, velký cca 1 000 000 komplexních čísel.

Jacobi Successive Over-relaxation

Test simuluje modifikovanou Jacobiho metodu. Procvičuje přístup do paměti při řešení diferenciálních rovnic. Provádí průměrování hodnot v poli, kde každá hodnota, je nahrazena průměrem okolních čtyř hodnot.

Test je prováděn na poli 100 x 100, respektive 1 000 x 1 000 u velkého vzorku.

Monte Carlo integration

Provádí aproximaci čísla π. Počítá integrál čtvrtkruhu $y = \sqrt{1 - x^2}$ na intervalu $[0, 1]$. Zvolí náhodný bod v jednotkovém čtverci a počítá poměr bodů uvnitř a vně kruhu. Test je zaměřen na generátor náhodných čísel, synchronizované volání funkcí a inlining funkcí.

Sparse matrix multiply

Používá rozptýlené dvourozměrné pole v řádkovém formátu. Test je zaměřen na nepřímé adresování a neregulární přístup k paměti. V každém řádku je přibližně pět nenulových čísel, která jsou rovnoměrně rozprostřena mezi první sloupec a hlavní diagonálu.

Velikost pole malého vzorku je 1 000 x 1 000 a obsahuje 5 000 nenulových čísel. Velký vzorek obsahuje 1 000 000 čísel v poli o rozměrech 100 000 x 100 000.

LU matrix factorization

Provádí LU dekompozici s částečnou pivotací. Testuje lineární algebru a operace s poli.

Operace je provedena na matici 100 x 100 při malém vzorku dat a na matici 1 000 x 1 000 při velkém vzorku dat.

2.4 Java Grande

Java Grande Forum Benchmark Suite je množina testů vyvinutých centrem EPCC (Edinburgh Parallel Computing Centre).

Jednotlivé testy jsou rozděleny do tří sekcí:

- Sekce 1: Nízko úrovňové operace
- Sekce 2: Jádra
- Sekce 3: Rozsáhlé aplikace

Cílem testů je poskytnou možnost změření a porovnání různých implementací java virtuálních strojů, nejen při vědeckých výpočtech, ale také například při provozu firemních databází nebo finančních simulací [4].

2.4.1 Sekce 1: Nízko úrovňové operace

Tato sekce dle předchozí klasifikace obsahuje sadu mikrobenchmarků. Provádí měření nízko úrovňových operací jako jsou aritmetické operace, volání metod a přetypování.

Každá operace je opakovaně prováděna v daném časovém úseku. Výsledky jsou poté udány jako počet operací za sekundu.

Arith

Benchmark měří výkonnost aritmetických operací (sčítání, násobení a dělení) nad primitivními datovými typy int, long, float a double. Výsledkem je počet operací za sekundu.

Assign

Měří čas, který je potřebný pro přiřazení proměnných různých typů. Proměnné můžou být skalární nebo prvek pole. A mohou být lokální, instanční nebo třídní. V případě instančních a třídních proměnných mohou náležet do stejné nebo jiné třídy. Výsledkem je počet přiřazení za sekundu.

Cast

Testuje přetypování mezi různými primitivními datovými typy. Provádí se přetypování int na float, int na double, long na float, long na double, plus zpětné přetypování double na int, double na long, float na int a float na long. Výsledkem testování je počet přetypování za sekundu.

Create

Test měří výkonnost vytváření objektů a polí. Vytváří se pole s prvky typu int, long, float. Výslednou jednotkou je počet vytvořených prvků za sekundu.

Loop

Měří režii při vykonávání cyklů. Testuje jednoduchý for cyklus, zpětný for cyklus a while cyklus. Výsledek je počet iterací za sekundu.

Math

Testuje výkon všech metod ve třídě java.lang.Math. Výsledkem testování je počet operací za sekundu.

Method

Měří režii volání metody. Metody mohou být instanční, instanční s modifikátorem final a třídní. Výsledkem měření je počet volání za sekundu.

Serial

Zjišťuje výkonnost serializace a deserializace objektu. Provádí zápis serializovaného objektu do souboru a čtení objektu ze souboru. Jednotkou měření je počet bytů za sekundu.

Exception

Měří výkon systému pro zpracování výjimek. Měří cenu vytvoření, vyhození a následného zachycení výjimky. Výsledkem měření je počet zpracovaných výjimek za sekundu.

2.4.2 Sekce 2: Jádra

Testy v této sekci vykonávají kratší kód, který provádí specifickou operaci. Tato operace bývá často použita v rozsáhlých aplikacích. Testy můžeme zařadit mezi mezobenchmarky.

Každý test obsahuje tři sady vstupních dat, malou, střední a velkou sadu.

Series

Počítá prvních N koeficientů Fourierovy řady na funkci $y = (x+1)^x$ na intervalu $[0, 2]$. Test je zaměřen na trigonometrické a transcendentní funkce. Výsledkem je počet koeficientů za sekundu.

Velikost N je dle velikosti vstupních dat 10 000, 100 000 nebo 1 000 000.

LUFact

Řeší soustavu lineárních rovnic, využitím LU faktorizace. Jedná se o Java verzi benchmarku Linpack. Zatěžuje paměť a výpočty s plovoucí řádovou čárkou.

Počet rovnic je podle velikosti vstupní sady.

HeapSort

Řadí pole celých čísel. Pro řazení je použit algoritmus heap sort. Test je zaměřený na práci s pamětí a celá čísla.

Počet čísel v poli je dán velikostí vstupních dat

Crypt

Provádí šifrování a dešifrování pole bytů. Pro šifrování je využit algoritmus IDEA (International Data Encryption Algorithm). Zaměřen na operace s bity a byty.

Velikost pole je dle velikosti vstupních dat 3 MB, 20 MB nebo 50 MB.

SOR, FFT, Sparse

Toto jsou převzaté testy z nástroje Scimark, který byl popsán výše, viz 2.3.

2.4.3 Sekce 3: Rozsáhlé aplikace

V této sekci provádí testy měření při vykonávání rozsáhlého kódu, jedná se o makrobenchmarky. Testy jsou vytvořeny modifikací reálných aplikací. Z aplikací jsou odstraněny I/O operace a grafické rozhraní.

Testy jsou navrženy k demonstraci potenciálu jazyka java při řešení reálných problémů.

Search

Tento test měří výkonnost při řešení hry čtyři v řadě, někdy také nazývané cestovní piškvorky nebo 2D piškvorky, na hrací desce 6 x 7 polí. Pro řešení je použit algoritmus alfa-beta.

Velikost testovaných dat je dána počátečním stavem hry. Zatěžuje práci s pamětí a celočíselné výpočty.

Euler

Řeší soustavu časově závislých Eulerových rovnic, pro simulaci proudění tekutiny v kanálu, kde jedna stěna kanálu obsahuje nerovnost, bouli. Pro aproximaci řešení se používá Runge-Kuttova metoda čtvrtého řádu.

Velikost kanálu je dána velikostí testovaných dat.

MD

Tento test modeluje interakci částic, neutrálních atomů nebo molekul podle Lennard-Jonesova modelu.

Počet částic je dán velikostí testovaných dat.

MC

Test provádí finanční simulaci. Metodou Monte Carlo stanovuje hodnotu finančního derivátu na základě podkladového aktivu, kterým můžou být například obligace nebo akcie.

Test generuje několik vzorků, které mají stejný průměr a fluktuaci hodnot jako historická data.

Počet vygenerovaných vzorků závisí na velikosti testovaných dat.

Ray Tracer

Test měří dobu vykreslení scény. Pro vykreslení je použita zobrazovací metoda sledování paprsků (raytracing), která slouží k vizualizaci 3D scény ve fotorealistické kvalitě. Scéna obsahuje 64 koulí.

Rozlišení výsledného obrazu je dána velikostí vstupních dat.

Kapitola 3

Balíček java.math

Jelikož je cílem práce porovnat výkonnost java virtuálních strojů při užití balíčku java.math, bude v následující kapitole, tento balíček krátce představen. Najdete zde popis tohoto balíčku. Následovaný popis třídy BigInteger, která je součástí tohoto balíčku. A nakonec bude popsána třída BigDecimal.

3.1 Popis

Balíček java.math obsahuje třídy pro výpočty s libovolnou přesností, přesnost výpočtů je tedy omezena pouze velikostí paměti hostitelského systému, na rozdíl od datových typů s omezeným rozsahem, jako jsou celočíselné datové typy short, int, long a datové typy s plovoucí řádovou čárkou float a double. [16, 14]. Pro celočíselné výpočty obsahuje třídu BigInteger, a pro desetinná čísla třídu BigDecimal.

3.2 BigInteger

BigInteger je neměnná třída pro celočíselné výpočty s libovolnou přesností. Umožňuje provádět běžné aritmetické operace, jako je sčítání, odčítání, násobení a dělení, dále obsahuje metody, které umožňují výpočet hodnot, jako metody nad primitivním datovým typem int v balíčku java.lang.Math, jedná se například o funkci pro výpočet maxima a minima dvou hodnot. Dále obsahuje funkce pro bitové operace a posuny. [11]

Chování aritmetických operací kopíruje chování při výpočtech aritmetických operací, které jsou definovány ve specifikaci jazyka Java. Například dělení nulou vyvolá výjimku ArithmeticException.

Při výpočtech binárních bitových operacích, jako jsou and, or nebo xor, je před vlastním výpočtem znaménkově rozšířen kratší z operandů. Chování kopíruje chování bitových logických operacích jazyka Java.

Bitové operace pracující s jedním bitem, jako jsou setBit, clearBit nebo flipBit, vykonávají operaci nad operandem, který je reprezentován ve dvojkové notaci. V případě že operand nemá požadovaný bit, je počet bitů rozšířen tak, aby mohla být operace vykonána. Žádná z těchto operací nemůže změnit znaménko operandu.

Dle [1] má neměnná třída, oproti proměnlivé, několik výhod:

- Lépe se navrhují, implementují a jsou méně náchylné k chybám. A jsou bezpečnější.

- Jsou jednoduší na použití, jelikož můžou být pouze v jednom stavu, a to tom, ve kterém byly vytvořeny.
- Jsou bezpečné při práci ve více vláknech a nevyžadují žádnou synchronizaci.
- Můžou být volně sdíleny, což může vést ke kešování, často používaných hodnot.
- Můžou také sdílen vnitřní reprezentaci, například při implementaci typu BigInteger, kde je číslo rozděleno na znaménko typu int a hodnotu, které je uložena v poli intů, může BigInteger při operaci negace, vytvořit nový objekt, který bude mít pouze jiné znaménko a hodnotu v poli bude sdílená.

Neměnné třídy mají taky jednu nevýhodu, jelikož potřebují vytvořit nový objekt pro každou novou hodnotu. Což může být velice nákladná operace, zejména u datového typu BigInteger, který může být velmi rozsáhlý. Například pokud bychom chtěli využít BigInteger pro udržení milionbitového čísla. Muselo by se, pokud bychom chtěli změnit hodnotu jednoho bitu pomocí metody flipBit, vytvořit nové milionbitové pole, které by se lišilo právě tím jedním bitem. Na rozdíl od této operace nad datovým typem BitSet z balíčku java.util, který není neměnný, a jediné co by mu stačilo pro vykonání operace, by bylo změnit jeden cílový bit.

3.3 BigDecimal

Stejně jako BigInteger je i BigDecimal neměnná třída. Tato třída slouží pro výpočty s desetinnými čísly s libovolnou přesností. Číslo BigDecimal se skládá z celočíselného základu s libovolnou přesností, které je reprezentováno předcházející třídou, a 32bitovou hodnotou, která určuje posun desetinné čárky. Pokud je hodnota kladná nebo nula, tak číslo udává počet cifer za desetinou čárkou. V případě kdy je hodnota záporná, číslo udává počet nul za základem čísla BigDecimal. Hodnotu čísla můžeme vypočítat jako $zaklad * 10^{-hodnota}$. Třída poskytuje metody pro aritmetické operace, zaokrouhlování, hešování a převod na ostatní datové typy [10].

Třída dává uživateli naprostou kontrolu nad chováním třídy při zaokrouhlování. V případě kdy uživatel nespecifikuje zaokrouhlovací mód a výsledek operace nemůže být reprezentován, jedná se například o čísla s neukončeným desetinným rozvojem, například 1/3, je vyvolána výjimka. V ostatních případech se operace provede po požadovanou přesnost, která je požadována uživatelem. Třída obsahuje celkem osm různých módů pro zaokrouhlování.

Pro jednu hodnotu existuje v třídě BigDecimal více reprezentací. Například číslo 100 může být reprezentováno základem 100 a hodnotou 0, nebo může být reprezentováno hodnotou 1 000 a hodnotou 1.

Všechnu aritmetické operace jsou prováděny tak, jako by byl prvně vypočítán přesný pomocný výsledek, který je následně zaokrouhlen podle zvoleného módu zaokrouhlování a požadované přesnosti. Zaokrouhlení probíhá pouze v případě pokud je to potřeba.

Jelikož je třída také neměnná, vztahují se na ní stejné výhody a nevýhody, které byly popsány u předešlé třídy BigInteger.

Kapitola 4

Návrh výkonnostních testů

V následující kapitole bude zdokumentovány nástrahy, které číhají při implementaci výkonnostních testů. Budou popsány obecné problémy, které mohou nastat při jakémkoliv testování a specifické problémy při testování java virtuálních strojů. Dále bude nástroj, který byl zvolen pro usnadnění eliminace těchto problémů. Nakonec jsou zdokumentovány jednotlivé testy, které jsou součásti aplikace na měření výkonnosti tříd BigInteger a BigDecimal.

4.1 Měření výsledků

Teoreticky je měření času pro vykonání operace velice jednoduché, stačí provést následující kroky:

1. Změřit počáteční čas
2. Provést operaci
3. Změřit konečný čas
4. Vypočítat rozdíl změřených časů

Dle [7, 2] ale měření není tak jednoduché. A výsledek měření může ovlivnit několik faktorů, které můžou značně zkreslit výsledek měření.

4.1.1 Eliminace mrtvého kódu

Velkým problémem při psaní benchmarků je eliminace mrtvého kódu. V některých případech může kompilátor vyhodnotit, že kód nijak neovlivňuje výsledek aplikace, a tak se rozhodne uvedenou část eliminovat. Problémem potom je, když kompilátor vyhodnotí a eliminuje část, případně celou měřenou operaci. Výsledek měření poté neodpovídá skutečné ceně vykonání dané operace a může vést k falešně krátkým časům pro vykonání dané operace.

Řešením této situace je poté nejen výsledek uložit do proměnné, ale hodnotu proměnné využít například pro výpis na obrazovku.

4.1.2 Vstupní data

Dalším problémem při měření výkonnosti operace, jsou špatně zvolená vstupní data. Například pokud by jsme měřili výkonnost funkce pro výpočet fibonnaciho posloupnosti, byl by výsledek měření jiný pro vypočítání prvního člena posloupnosti oproti stému členu posloupnosti.

Z tohoto důvodu musí být vstupní data proměnná. Další důvod, aby byla data proměnná je, že chytrý kompilátor může vyhodnotit, že výsledek operace je neměnný a nahradit výpočet výsledku, pouze tímto výsledkem.

Důležité je i zvolit vhodný rozsah hodnot vstupních dat. U implementace fibannaciho posloupnosti může být například definován rozsah vstupních hodnot, pro která má smysl výsledek počítat. Například rozsah $[0, 46]$, kde fibonnaci(46) je největší číslo, které může být uloženo v datovém typu int. V ostatních případech funkce vyhodí výjimku. Pokud by bylo pro generování použit generátor náhodných čísel bez omezení rozsahu, výsledky měření by byly zkreslené a neodpovídaly by běžnému použití této funkce.

4.1.3 Irelevantní operace

Toto může souviset s předchozím problémem, pokud se budou data generovat v měřeném úseku, bude do výsledku započítán i čas, který je k tomu potřeba. Z tohoto důvodu by měla být vstupní data vygenerována před vlastním měřením.

S měřením je ale vždycky spojena nějaká režie, ať již třeba jenom režie pro zavolání funkce pro získání času a přiřazení tohoto času do proměnné.

4.1.4 Mixed mode

Tento problém je specifický pro java virtuální stroje, které mají komplikovaný životní cyklus. Cílem měření výkonnosti java virtuálních strojů je porovnat výkon v ustáleném stavu, který se značně liší od prvotního výkonu.

Je to dáno tím, že moderní java virtuální stroje typicky ze začátku kód provádí čistě interpretovaný, během toho shromažďují profilové informace, a poté provedou JIT (just-in-time) kompilaci. Než se tedy dosáhne ustáleného stavu musí být operace již několikrát provedena.

První provedení je často výrazně pomalejší, než ostatní. Je to dáno tím, že při prvním provádění se musí načíst požadované třídy, což obvykle zahrnuje čtení z disku, parsování a verifikaci. Po prvním provedení tedy nastane výrazné zlepšení výkonu. Poté se dochází ke zlepšení v několika krocích, než se dosáhne ustáleného výkonu.

Pro eliminaci tohoto problému bývá do testování zahrnuta tzv. warmup-perioda, při které je testovaný úsek několikrát proveden, aby měl java virtuální stroj dostatek času, pro nasbírání profilových dat a provedení JIT kompilací. Po provedení by měl být výsledek měření v ustáleném stavu.

Problémem při testování je i to, že interpret může jinak vyhodnotit profilové informace získané při izolovaném použití při vykonávání jedné operace, než při vykonávání v reálné aplikaci. Kompilátor tak může provést jiné optimalizace, než které by použil při skutečném nasazení. Tomuto problému se nejde úplně vyhnout, leda by se měřil výkon skutečné aplikace, čímž by se ale zvětšil podíl irelevantních operací.

4.2 Použité nástroje

Z předešle uvedených důvodů jsem se rozhodl pro implementaci využít knihovnu od OpenJDK JMH (Java Microbenchmarking Harness), která výrazně usnadňuje vývoj mikrobenchmarků. Při správném použití odstraní problém eliminace mrtvého kódu, dále umožňuje eliminovat problém, který může vzniknout při skládání konstantních proměnných. Dále umožňuje spustit benchmark v různých módech, umožňuje měřit průměrný čas, distribuci jednotlivých časů nebo čas pro vykonání jedné operace [8].

4.3 Testy

Jednotlivé testy jsou rozděleny do čtyř modulů podle počtu operandů a typu operandů.

V souboru BigIntegerSingleBench jsou implementovány testy, které testují výkonnost datového typu BigInteger, při vykonávání operací, které nevyžadují druhý operand typu BigInteger. Jsou zde implementovány funkce pro měření výkonu například funkcí abs, flipBit, setBit, pow...

V souboru BigIntegerDoubleBench jsou pak implementovány testy, které testují metody, které vyžadují druhý operand typu BigInteger. Jedná se především o aritmetické operace a metody provádějící logické operace.

Názvy jednotlivých testů předcházející modulů odpovídají názvům metod z třídy BigInteger, které testují, u jednotlivých tříd můžete nastavit z kolika bitovými operandy se budou operace provádět.

Ve třetím modulu BigDecimalSingleBench najdete implementaci testů, které měří výkonnost datového typu BigDecimal při unárních operacích, jako je například negace.

V posledním modulu BigDecimalDoubleBench jsou implementace testů, které měří výkonnost převážně aritmetických operací, výjimkou jsou jenom testy max a min, které měří výkonnost uvedených metod.

Názvy metod z modulu BigDecimalSingleBench a BigDecimalDoubleBench pak odpovídají názvům metod z třídy BigDecimal. Parametry můžete určit počet bitů základu, který ovlivňuje přesnost daného čísla, viz 3.3.

Vstupní data pro jednotlivé testy jsou generována ve zvláštní třídě, která je k tomu určená.

Kapitola 5

Interpretace výsledků

V této kapitole bude zdokumentováno použité počítačové vybavení, které bylo použito pro získání testovaných výsledků. Dále budou představeny jednotlivé java virtuální stroje, které byly testovány. Poté bude následovat vlastní porovnání těchto strojů na základě výsledku získaných použitím nástroje představeného v předchozí kapitole. Java virtuální stroje budou nejprve porovnány ve výkonnosti při vykonávání aritmetických operací nad datovým typem BigInteger. Dále budou porovnány při vykonávání logických operací nad datovým typem BigInteger. V neposlední řadě budou implementace porovnány v aritmetických operacích nad datovým typem BigDecimal.

5.1 Testovací sestava

Jelikož cílem této práce je porovnat výkonnost java virtuálních strojů při užití tříd z balíčku java.math a nejde o porovnání výkonnostních vlastností použitého hardware, uvádím zejména z důvodu reprodukovatelnosti testů základní popis komponent testovacího počítače.

Testování probíhalo na počítači Asus K55A-SX198. Verze BIOSu od společnosti American Megatrends je uvedena K55A.305 a pochází z 24. května 2012. Počítač je vybaven procesorem Intel Pentium B970, 64bitová architektura, 2 jádra o frekvenci 2,30 GHz. Procesor disponuje vyrovnávací pamětí o velikosti 128 Kb na úrovni L1, 512 Kb na úrovni L2 a 2 Mb na úrovni L3. V počítači je nainstalován SODIMM DDR3 paměťový modul s kapacitou 4 GB, pracující na frekvenci 1 333 MHz.

Testování probíhalo pod operačním systémem OpenSUSE verze 13.1.10. Verze linuxového jádra je 3.11.10. Před vlastním testováním byly zastaveny všechny nepotřebné procesy a naplánované aplikace. Dále byla vypnuta správa napájení a pomocí nástroje cpufreq-set [3] nastaven regulátor frekvence procesoru na performance.

Aby bylo dosaženo co nejlepších podmínek pro testování, přestože probíhalo na přenosném počítači, byl počítač po celou dobu testování připojen do elektrické sítě.

5.2 Testované metody

Přestože bylo implementováno testování i dalších metod z tříd BigInteger a BigDecimal. Rozhodl jsem se provést testování pouze těchto základních metod:

- BigInteger.add

- BigInteger.subtract
- BigInteger.multiply
- BigInteger.divide
- BigDecimal.add
- BigDecimal.subtract
- BigDecimal.multiply
- BigDecimal.divide
- BigInteger.add
- BigDecimal.andNot
- BigDecimal.or
- BigDecimal.xor

Výběr metod vychází z předpokladu, že aritmetické funkce jsou nejčastěji volané funkce těchto tříd, k tomu byly přidány všechny binární logické operace třídy BigInteger.

5.3 Testované JVM

Pro testování byly vybrány následující implementace java virtuálních strojů:

- Oracle Java SE 6 [12], bližší informace o použité verzi najdete na obrázku 5.1.

```
java version "1.6.0_37"
Java(TM) SE Runtime Environment (build 1.6.0_37-b06)
Java HotSpot(TM) 64-Bit Server VM (build 20.12-b01, mixed mode)
```

Obrázek 5.1: Výpis příkazu java -version

- Oracle Java SE 7 [13], v době započatí práce se jednalo o aktuální verzi, informace o verzi jsou na obrázku 5.2

```
java version "1.7.0_45"
Java(TM) SE Runtime Environment (build 1.7.0_45-b18)
Java HotSpot(TM) 64-Bit Server VM (build 24.45-b08, mixed mode)
```

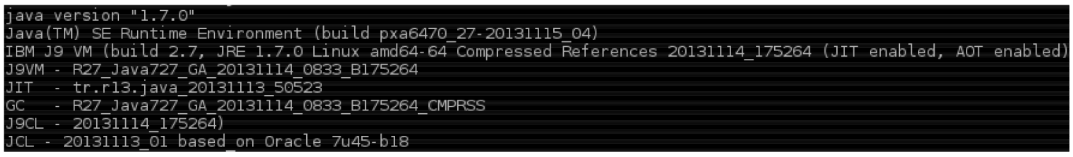
Obrázek 5.2: Výpis příkazu java -version

- OpenJDK 7 [9], jedná se o open-source verzi java virtuálního stroje, na obrázku 5.3 najdete bližší informace o verzi, získané výpisem příkazu java -version.

```
java version "1.7.0_51"
OpenJDK Runtime Environment (IcedTea 2.4.4) (suse-24.13.5-x86_64)
OpenJDK 64-Bit Server VM (build 24.45-b08, mixed mode)
```

Obrázek 5.3: Výpis příkazu java -version

- IBM Java SE Version 7 [5], posledním testovaným strojem je implementace od firmy IBM, jedná se o poslední vydanou verzi. Informace o verzi jsou na obrázku 5.4.



Obrázek 5.4: Výpis příkazu java -version

5.4 Výsledky

5.4.1 BigInteger: Aritmetické operace

Sčítání

Na obrázku 5.5 můžeme vidět výsledek testování operace BigInteger.add(). V prvním sloupci, ve kterém jsou zobrazeny výsledky měření při výpočtech s 32bitovými operandy, které mohou být vnitřně reprezentovány primitivním datovým typem, můžeme vidět vyrovnané výsledky implementací OpenJDK a Oracle Java verze 1.7. Dále můžeme vidět zlepšení výkonu oproti předcházející verzi 1.6. Nejlépe z testovaných implementací v případě 32 bitových operandů vychází IBM java.

Při ostatních kombinacích operandů je ale výsledek virtuálního stroje IBM přibližně o 50 procent horší, než výsledek od Oracle java verze 1.7, případně od OpenJDK. Výjimku tvoří operace při kterých je počet bitů operandů vyrovnaný. V těchto případech je IBM o něco výkonnější.

Mezi výsledky implementace od Oracle verze 1.7 a OpenJDK najdeme minimální rozdíly, obě implementace jsou svými výsledky vyrovnané. Při bedlivém zkoumání můžeme vidět mezi nimi drobné nuance, které jsou ale tak malé, že mohou být způsobené pouze chybou měření.

Při porovnání předchozí verze implementace od společnosti Oracle s novější verzi 1.7, je u všech kombinací operandů pozorovatelné zlepšení výkonu, které se pohybuje kolem 10 procent.

Z grafu je taky patrné, že pořadí operandů nemá vliv na výsledný čas k provedení operace. Což se dalo předpokládat vzhledem ke komutativnosti operace sčítání.

Pokud uděláme aritmetický průměr trvání operace sčítání při všech měřených kombinací operandů, zjistíme, že na testované sestavě trvá sečtení přibližně 203 ns na virtuálních strojích OpenJDK a Oracle verze 1.7. Což je o 22 ns lepší, než výsledek předchozí verze 1.6, které výpočet trvá 225 ns. IBM zaostává přibližně 70ns s výsledkem 274 ns.

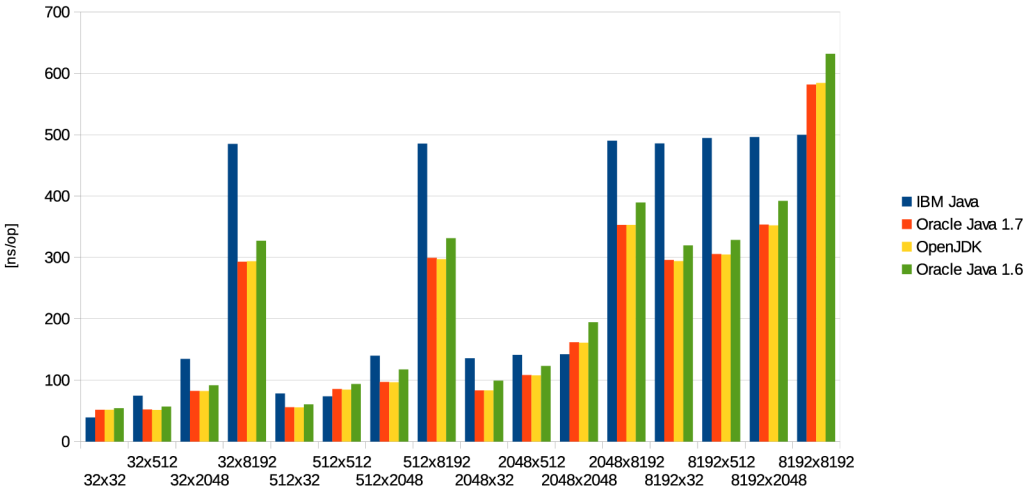
Odčítání

Na obrázku 5.6 můžeme vidět graf operace odčítání BigInteger.subtract(). Jak na znázorněném grafu vidět výsledek operace odčítání je velmi podobný, předcházející operaci sčítání.

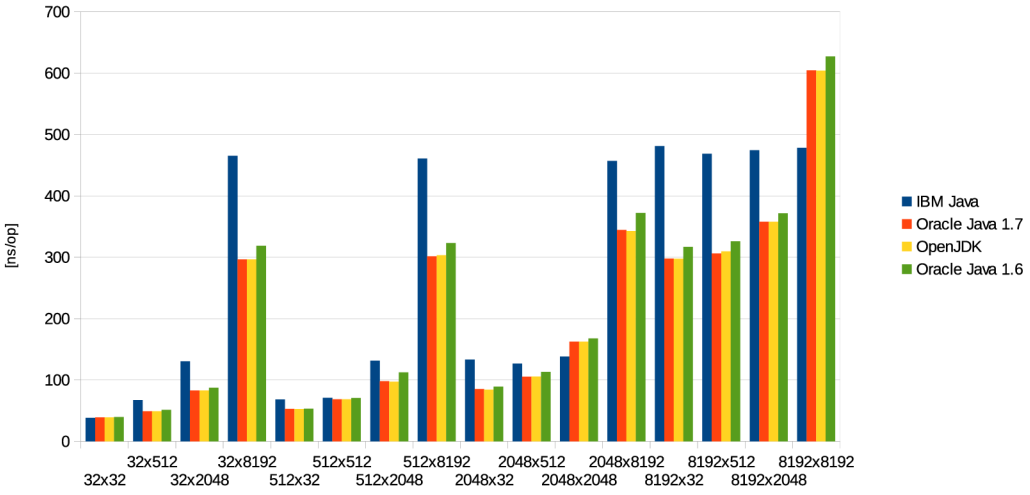
U operace odčítání je o něco menší rozdíl mezi implementacemi IBM a Oracle. U operace odčítání, také není vidět znatelný rozdíl při počítání s 32 bitovými operandy a všechny implementace jsou si víceméně rovny.

Ani u operace odčítání nezáleží na pořadí operandů, alespoň co se týká výkonnosti.

Menší rozdíly mezi implementacemi potvrzují i průměrné hodnoty, kde přestože operace na strojích Oracle 1.7 a OpenJDK trvá stejnou dobu jako sčítání 203 ns. Tak implementace od IBM a Oracle 1.6 získaly 10 ns a stroj od IBM zaostávají už jen o 60 ns s výsledkem 261 ns. A předchozí verze Oracle 1.6 s 214 ns dokonce zmenšila rozdíl na polovinu na 11 ns.



Obrázek 5.5: BigInteger.add()



Obrázek 5.6: BigInteger.subtract()

Násobení

Výsledky měření operace BigInteger.multiply(), násobení dvou čísel typu BigInteger, můžete vidět na obrázku 5.7. Graf je vzhledem k rozptylu hodnot rozdělen na dvě části.

Při porovnání naměřených hodnot u implementací od společnosti Oracle můžeme vidět drobné zlepšení oproti předcházející verzi, s výjimkou posledního sloupce, který znázorňuje operaci s operandy, které mají 8 192 bitů. V tomto případě jsou výsledky starší verze o trochu lepší. Jedná se přibližně o 3 000 ns, starší verzi operace trvá cca 132 000 ns, zatímco nové verzi to trvá 135 000 ns.

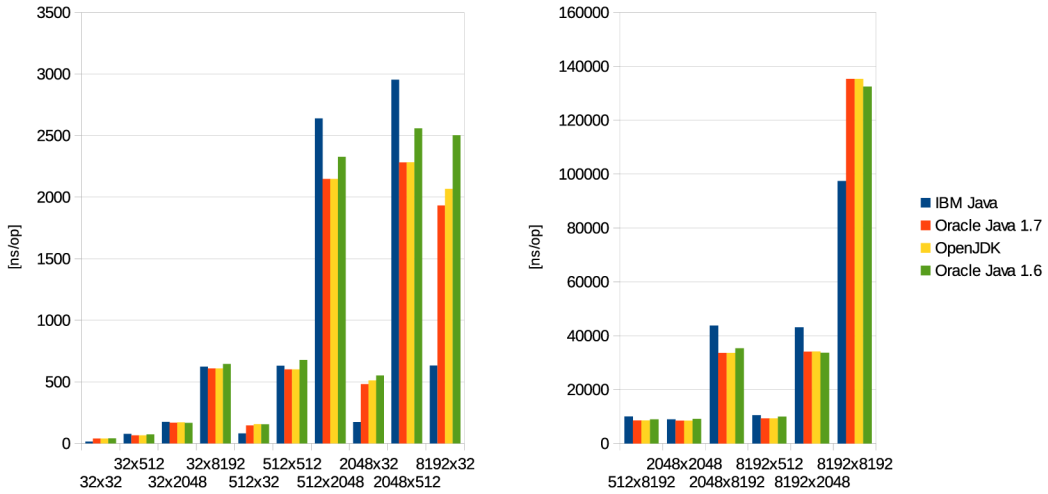
Výkonnost OpenJDK a Oracle 1.7, jak je vidět na grafu, se prakticky neliší. Průměr hodnot, který se pohybuje kolem 14 860 ns, se liší pouze o 6 ns.

Výkon interpretu od společnosti IBM při násobení za ostatními implementacemi mírně pokulhává. Výjimkou je pouze násobení, kdy první operand má velikost 2 048 nebo 8 192 bitů a druhý je 32bitový. A druhou výjimku tvoří případ, kdy jsou oba operandy 8 192bitové.

První případ je také zajímavý tím, že v případě prohození operandů, tzn. že první je 32bitový a druhý má 2 048 nebo 8 192 bitů, se výrazně zlepší výkonnost virtuálních strojů od společnosti Oracle a OpenJDK. Bohužel jsem pro tuto anomálii nenašel vysvětlení.

Ostatní kombinace operandů, se projevují podle předpokladu, a čas pro provedení násobení se při prohození operandů neliší.

Přestože IBM u většiny kombinací argumentů je výkonnostně za ostatními. V průměru časů na provedení jednotlivých operací je se svými 13 854 ns na prvním místě. Na druhém a třetím místě jsou s cca 14 860 interpreti OpenJDK a Oracle 1.7. Na posledním místě je pak předchozí verze Oracle 1.6 s 14 950 ns. Úspěch IBM je hlavně dán výrazně lepším výsledkem při násobení velkých 8 192bitových čísel. Pokud by do tohoto průměru nebyl započítán, IBM by skončilo na posledním místě, ostatní pořadí by se nezměnilo. Průměr na operaci by činil 8 281 ns u IBM oproti 6 830 ns u OpenJDK a Oracle 1.7.



Obrázek 5.7: BigInteger.multiply()

Dělení

Obrázek 5.8 zobrazuje výsledky operace dělení BigInteger.divide(). Ze stejného důvodu, jako u operace násobení, je graf rozdělen do dvou částí s různými měřítky.

V levé části grafu můžete vidět výsledek měření, kdy první operand metody, dělenec, má stejný nebo menší počet bitů než druhý operand, dělitel. Výjimku tvoří pouze případ, kdy dělitel má 32 bitů a dělenec 512 bitů.

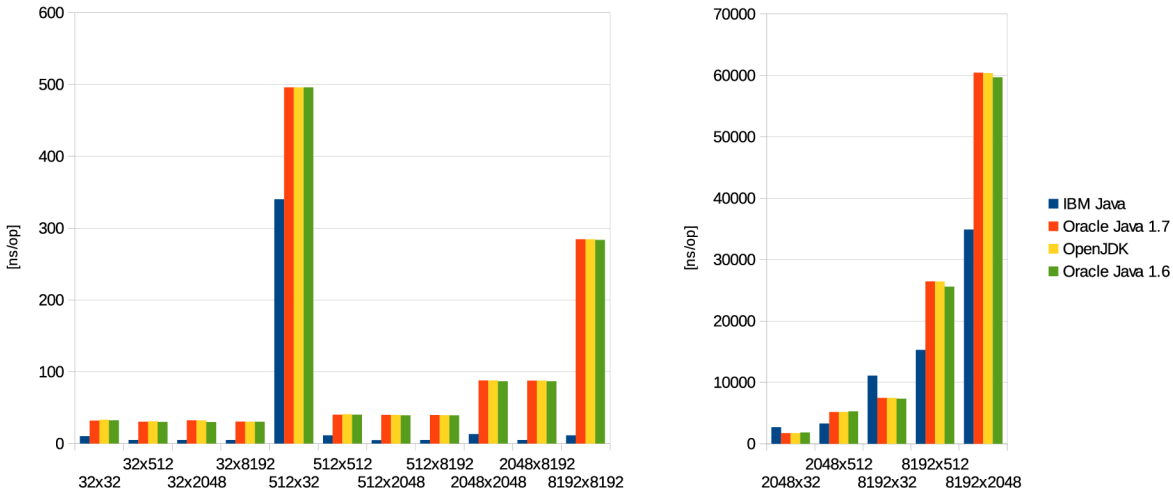
Jak si můžete všimnou jsou výsledné časy téměř totožné, u interpretu IBM všechny, u ostatních interpretů můžeme vidět nárůst, když má dělenec 2 048 bitů a následně 8 192 bitů. Vyrovnanost výsledků je dána tím, že výsledkem celočíselného dělení, kdy dělitel má stejný řád jako dělenec, případně větší, je 0, výjimečně 1. Nejlépe si s tímto problémem poradil virtuální stroj od společnosti IBM. Ostatní interpreti jsou na tom víceméně stejně. Když vezmeme v úvahu charakter počítaného příkladu je zajímavý nárůst času pro výpočet, který je vidět v posledních třech sloupcích v levé části.

V pravé části a na sloupci, kdy má dělenec 512 bitů a dělitel 32 bitů, pak můžeme vidět srovnání jednotlivých strojů, při opravdovém složitém výpočtu.

Při porovnání můžeme vidět, že implementace od IBM je při dělení skoro o polovinu rychlejší než ostatní implementace. Nejvíce je toto pozorovatelné v posledním sloupci, kde dělitel má 2 048 bitů a dělenec 8 192 bitů.

Ostatní implementace jsou v tomto případě velmi vyrovnané, a nelze ani nalézt rozdíl mezi novou a starší verzi od společnosti Oracle.

Jelikož by mohli být výsledky zkreslené, kdyby se výpočet průměru času pro provedení operace dělení počítal z případů, kdy je dělitel výrazně větší než dělenec, byly tyto případy z výpočtu odstraněny. Po provedení průměru vychází průměrný čas na výpočet přibližně 11 000 ns pro IBM, a kolem 17 000 pro ostatní implementace.



Obrázek 5.8: BigInteger.divide()

Souhrn

Z výsledku předchozích operací byl sestaven kompozitní výsledek, který vznikl zprůměrováním časů pro vykonání jednotlivých operací. Výsledek můžete vidět na obrázku 5.9.

Jak je z uvedeného grafu vidět, tak v případě rovnoměrného provádění všech operací, vychází nejlépe pro java virtuální stroj vyvinutý společností IBM, která je skoro o třetinu rychlejší. Ale jak bylo vysvětleno v předcházející části velmi záleží na způsobu použití. Implementace OpenJDK je rovnocenným konkurentem interpretu od společnosti Oracle verze 1.7 a výsledky předchozích testů byly téměř totožné.

I když v předcházejících testech bylo vidět zlepšení výkonu nové verze 1.7 oproti starší verzi 1.6, ve výsledku není tolik výrazné.

Obrázek 5.9: Kompozitní výsledek pro aritmetické operace datového typu BigInteger

5.4.2 BigInteger: Logické operace

Operátor and

Na obrázku 5.10 je zobrazen graf, který ukazuje výsledky logické operace and.

Při porovnání výsledků si můžeme všimnou, že jsou velmi podobné s operacemi sčítání a odčítání. Jedinou výjimku tvoří výsledky od IBM.

Při porovnání výsledku vidíme, že čas pro vykonání logické operace and u implementace od IBM, je téměř 10 krát rychlejší.

Dále si můžeme všimnou, že na rozdíl od předchozích testů, nedošlo ke zlepšení výkonnosti novější verze od společnosti Oracle oproti předcházející. Naopak můžeme sledovat mírné zhoršení.

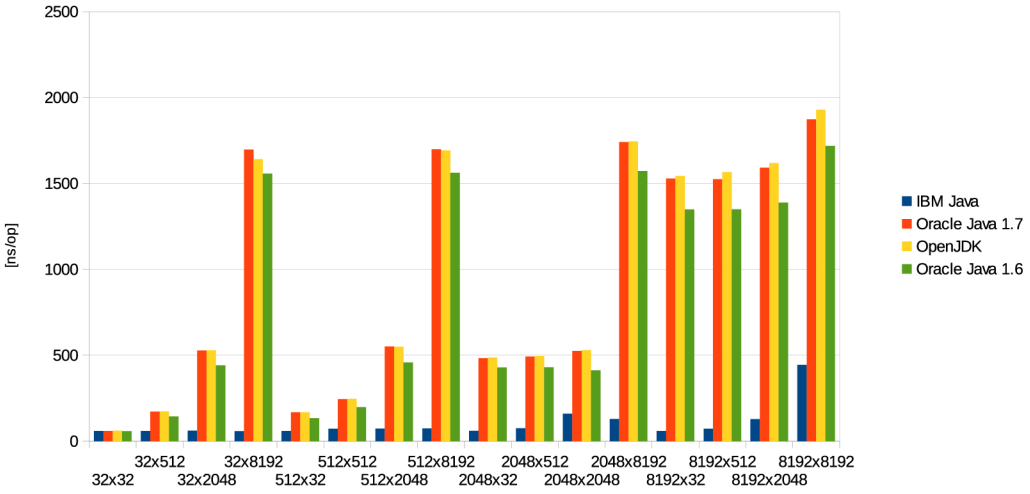
Implementace od OpenJDK a Oracle 1.7 jsou, tak jako v předcházejících testech velmi vyrovnané, ale můžeme vidět drobné zaváhání u implementace OpenJDK.

Průměrný čas pro vykonání operace byl 103 ns, 930 ns, 936 ns a 825 ns v tomto pořadí IBM, Oracle 1.7, OpenJDK a Oracle 1.6.

Operátor andNot

Výsledky měření operace BigInteger.andNot(), můžete vidět na obrázku 5.11.

Výsledek je téměř totožný s předcházející operací. Největší rozdíl je vidět v posledních sloupcích u společnosti IBM, kde vidíme vzestup času pro vykonání operace andNot oproti



Obrázek 5.10: BigInteger.and()

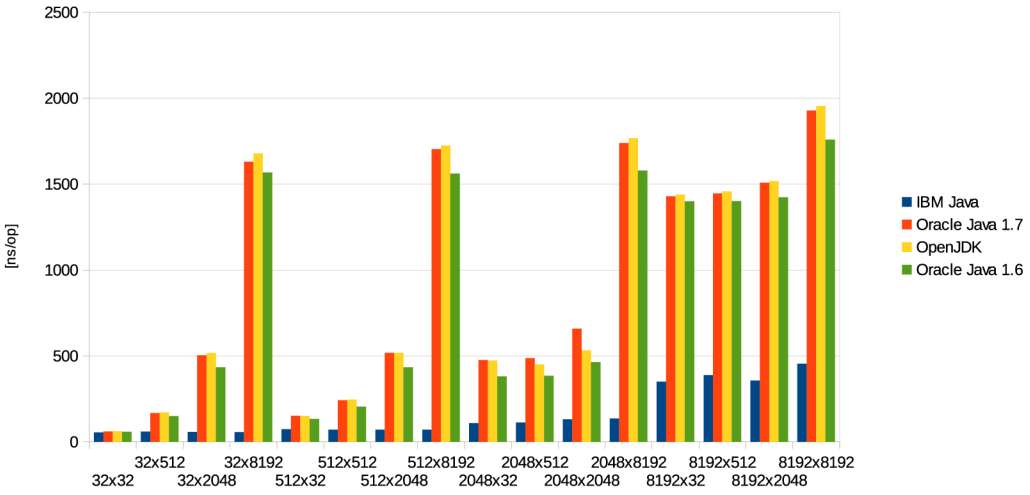
operaci and.

V porovnání s ostatními stroji vychází měření znovu výrazně lépe pro IBM.

Jediný případ, kdy jsou výsledky operace vyrovnané je v případě, kdy jsou oba operandy operace 32bitové.

Ani v případě operace andNot nedošlo ke zlepšení výkonu novější verze Oracle oproti předcházející verzi.

Průměrné časy jsou téměř totožné s operací and, z tohoto důvodu nejsou do této práce zahrnuty.



Obrázek 5.11: BigInteger.andNot()

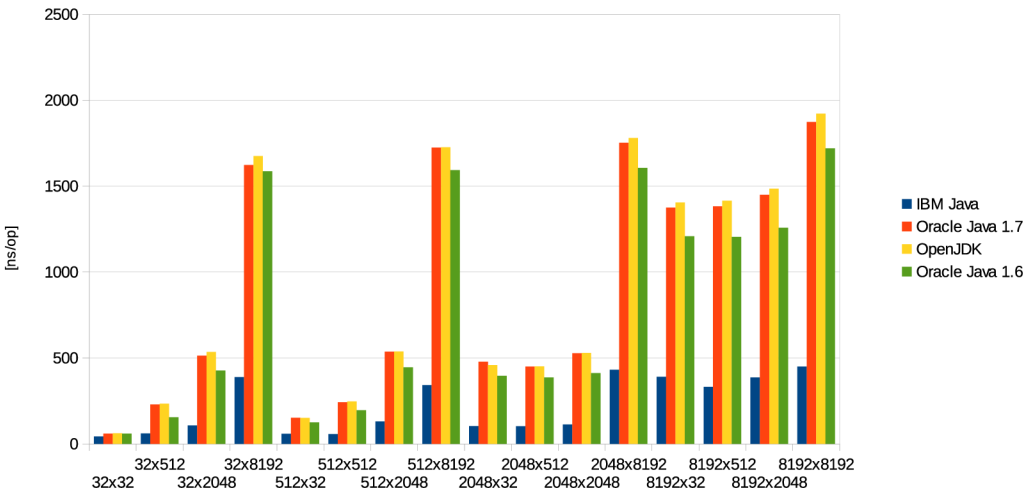
Operátor or a xor

Na obrázcích 5.12 a 5.13 jsou zobrazeny výsledky operací BigInteger.or() a BigInteger.xor().

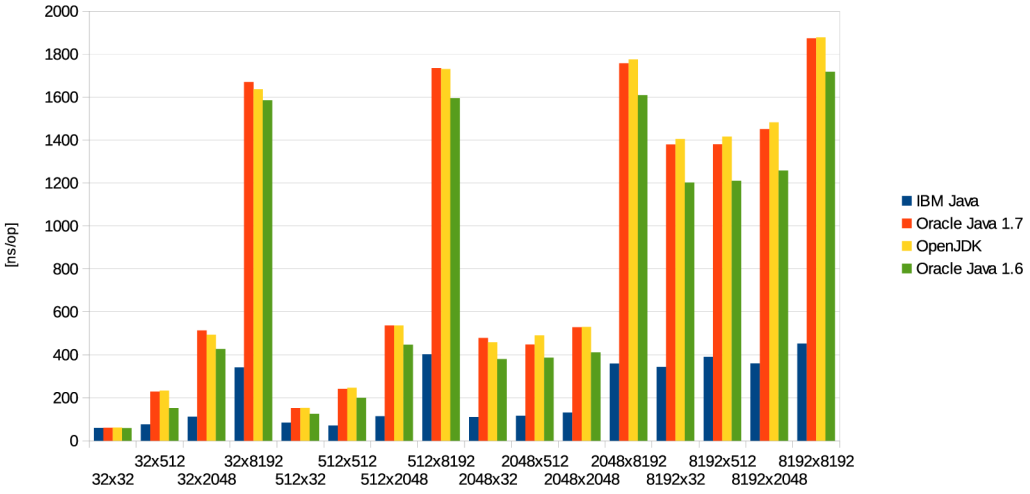
Při porovnání těchto grafů můžeme vidět, jak si jsou uvedené funkce výkonnostně podobné. Výsledky jsou podobné i s předcházejícími operacemi and a andNot. S tím rozdílem, že operace and a andNot byly o něco rychlejší.

Před ostatními implementacemi opět vyniká implementace od IBM, jako v případě předchozích logických operací.

OpenJDK mírně zaostává za implementací od Oracle verze 1.7., Oracle 1.6 překvapivě předčil v této operaci novější verzi 1.7, která si naopak trochu pohoršila.



Obrázek 5.12: BigInteger.or()



Obrázek 5.13: BigInteger.xor()

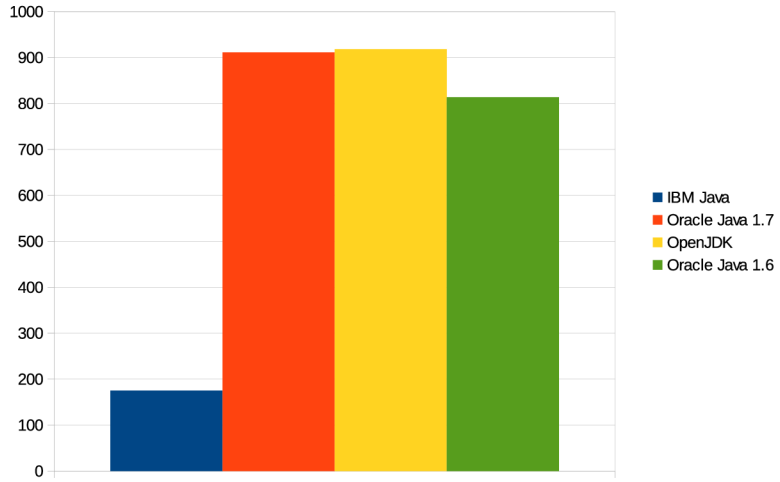
Souhrn

Na obrázku 5.14 můžete pozorovat kompozitní výsledek logických operací, který je získán zprůměrováním hodnot pro vykonání jednotlivých operací.

Vzhledem k předešle uvedeným výsledkům není překvapivé, že nejlépe se umístil interpret od společnosti IBM, který se svým průměrem 175 ns na operaci výrazně předčil své konkurenty.

Na druhém místě skončila překvapivě starší implementace Oracle verze 1.6, které trvalo vykonnání logické operace průměrně 814 ns.

Na třetím a čtvrtém místě pak skončili s těsným rozdílem implementace od společnosti Oracle verze 1.7 a OpenJDK. Operace trval virtuálnímu stroji Oracle 1.7 911 ns a virtuálnímu stroji OpenJDK 918 ns.



Obrázek 5.14: Kompozitní výsledek pro logické operace datového typu BigInteger

5.4.3 BigDecimal: Aritmetické operace

Sčítání

Na obrázku 5.15 můžeme vidět výsledek testování operace BigDecimal.add().

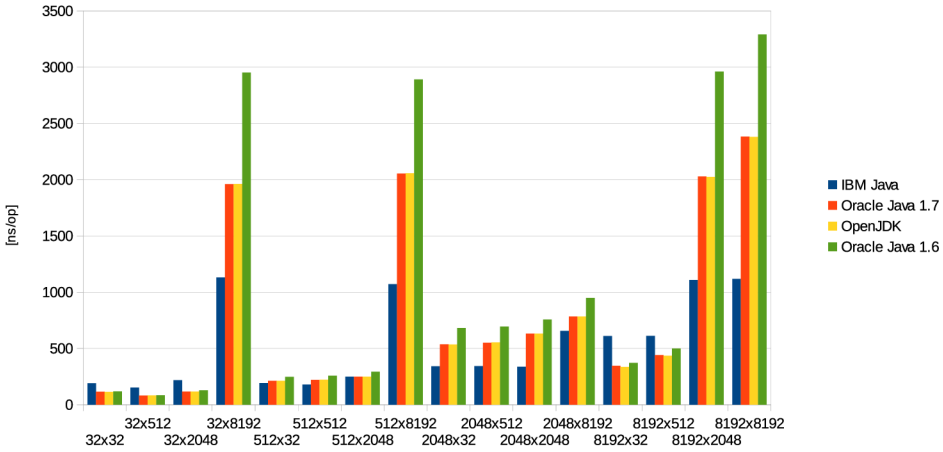
Při porovnání s operací sčítání u datového typu BigInteger můžeme vidět, že výsledky nejsou tak vyrovnané a mezi jednotlivými implementacemi jsou mnohem větší rozdíly.

Dobře si vede stroj od IBM, hlavně při výpočtech s operandy, které mají více bitů. Jak je ale vidět v prvních třech sloupcích, tak při výpočtech s málo bitovými operandy je implementace od IBM v porovnání s ostatními o polovinu horší. Celkově je ale průměr vykonání operace o 262 ns rychlejší než u implementace u OpenJDK.

Na grafu je také patrné výrazné zlepšení novější verze od společnosti Oracle. Které je v průměru o 279 ns rychlejší.

Implementace od OpenJDK a Oracle verze 1.7 se ve výsledcích téměř neliší, rozdíl průměru hodnot pro vykonání jedné operace není ani jedna ns, která se vejde do chyby měření.

Průměr času pro vykonání operace sčítání činí u stroje IBM 532 ns. U strojů OpenJDK a Oracle 1.7 to je přibližně 794 ns. U starší verze Oracle 1.6 operace průměrně trvá 1073 ns.



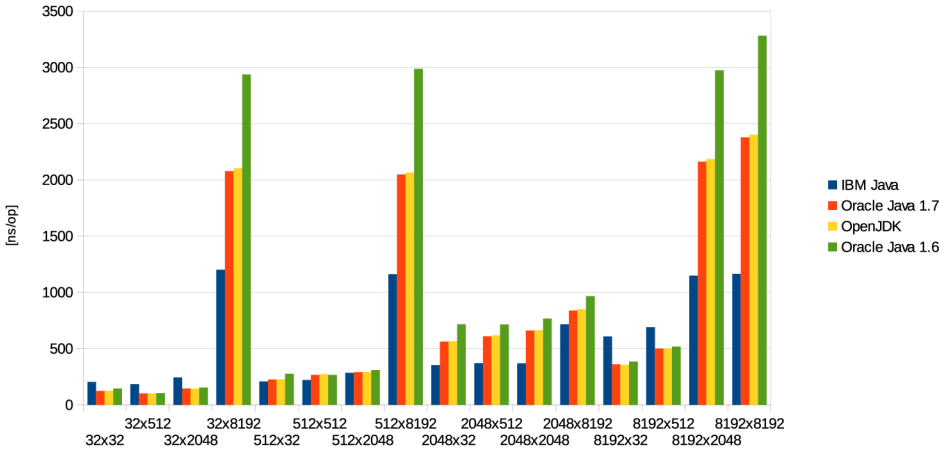
Obrázek 5.15: BigDecimal.add()

Odčítání

Na obrázku 5.16 jsou zobrazeny výsledky získané měřením výkonnosti operace BigDecimal.subtract().

Výsledky jsou téměř totožné jako u operace odčítání. Najdeme zde větší rozdíl průměrné doby vykonání operace mezi implementacemi OpenJDK a Oracle 1.7, který je 7 ns.

Také se u všech implementací mírně zvýšila doba, které je potřebná pro vykonání operace. U interpretu od IBM trvá operace v průměru 570 ns oproti 532 ns u sčítání, u Oracle 1.7 834 ns a u OpenJDK 841 ns oproti předešlým 794 ns a u Oracle 1.6 1093 ns oproti 1073 ns.



Obrázek 5.16: BigDecimal.subtract()

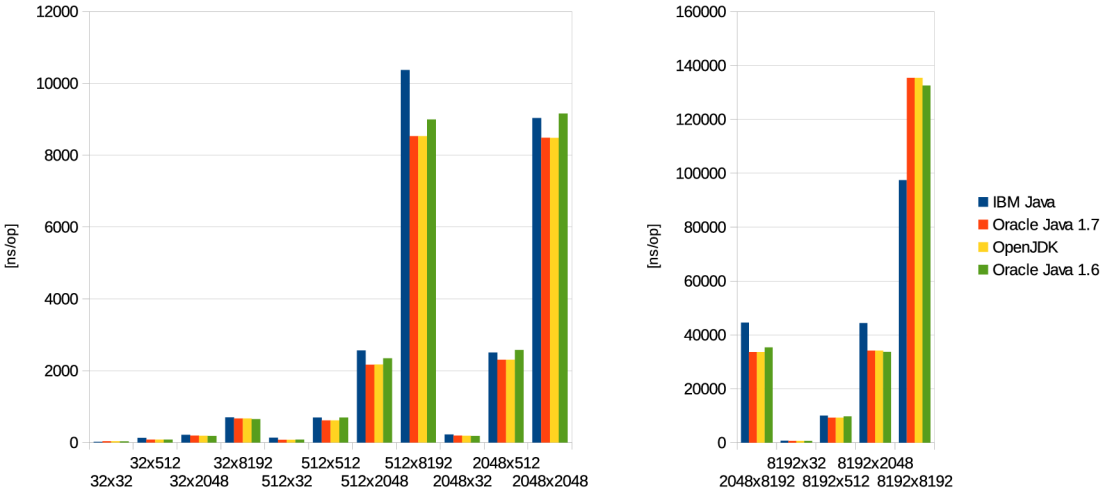
Násobení

Výsledky měření operace násobení najdete na obrázku 5.17. Graf je z vdůvodu rozptylu času, který je potřeba pro vykonání operace, rozdělen do dvou částí s různým měřítkem na ose y.

Pokud porovnáme měření java virtuálního stroje od společnosti IBM s ostatními implementacemi, můžeme vidět, že kromě poslední operace s operandy, jejichž základ je tvořen 8192bitovými operandy, jsou výsledné časy pro vykonání operace větší než u ostatních strojů.

Mezi výsledky implementace od Oracle verze 1.7 a OpenJDK najdeme minimální rozdíly, obě implementace jsou svými výsledky vyrovnané. Předchozí verze Oracle 1.6 je u většiny kombinací operandů o několik ns pomalejší, i když v posledním sloupci je skoro o 3 000 ns rychlejší.

Průměr času pro vykonání operace násobení činí u stroje IBM 13 981 ns. U strojů OpenJDK a Oracle 1.7 to je přibližně 14 778 ns. U starší verze Oracle 1.6 operace průměrně trvá 14 808 ns.



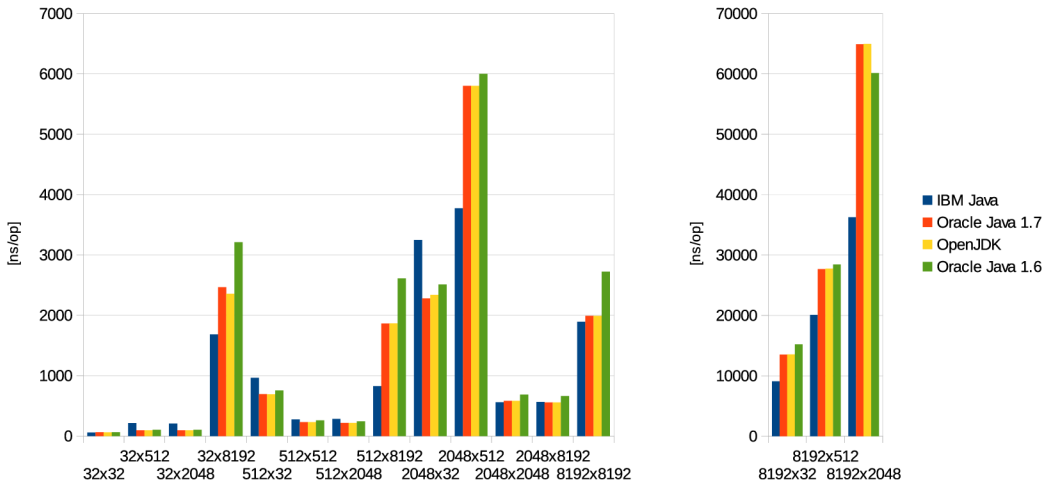
Obrázek 5.17: BigDecimal.multiply()

Dělení

Obrázek 5.18 zobrazuje výsledky operace dělení BigDecimal.divide(). Stejně jako u operace násobení je graf, z důvodu rozsahu naměřených časů, rozdělen do dvou částí s různými měřítky.

Na grafu je vidět, že nejlépe si vede virtuální stroj od IBM, nejvíce je to pozorovatelné v posledním sloupci, kde čas pro vykonání operace je o více jak 1/3 menší.

V průměru časů na provedení operace dělení je se svými 4 998 ns na prvním místě IBM Java. Na druhém a třetím místě jsou s cca 7 692 interpreti OpenJDK a Oracle 1.7. Na posledním místě je pak předchozí verze Oracle 1.6 s 7 733 ns.



Obrázek 5.18: BigDecimal.divide()

Souhrn

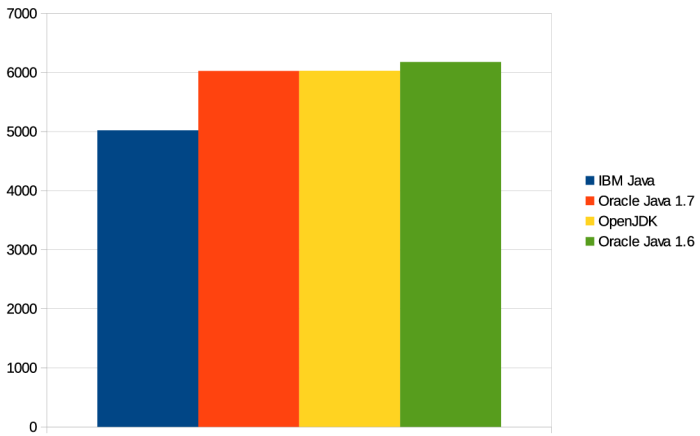
Z výsledku předchozích operací nad datovým typem BigDecimal byl sestaven kompozitní výsledek, který vznikl zprůměrováním časů pro vykonání jednotlivých operací. Výsledek můžete vidět na obrázku 5.9.

Jak je z uvedeného grafu vidět, tak v případě rovnoměrného provádění těchto operací, bylo nejlepšího výsledku dosaženo java virtuálním strojem vyvinutým společností IBM.

Výsledek měření implementace OpenJDK je téměř totožný s Oracle verze 1.7. Průměr času pro vykonání všech operací se liší o pouhé 2 ns.

Na grafu také vidíme, že novější implementace Oracle oproti předcházející verzi získala, při vykonávání několik ns. A je vidět posun výkonu dopředu.

Průměrný čas pro vykonání aritmetické operace je 5020 ns u IBM, 6024 ns Oracle 1.7, 6026 ns OpenJDK a 6177 ns Oracle 1.6.



Obrázek 5.19: Kompozitní výsledek pro aritmetické operace datového typu BigDecimal

5.4.4 Shrnutí

Na obrázku 5.20 můžete vidět kompozitní výsledek celého měření.

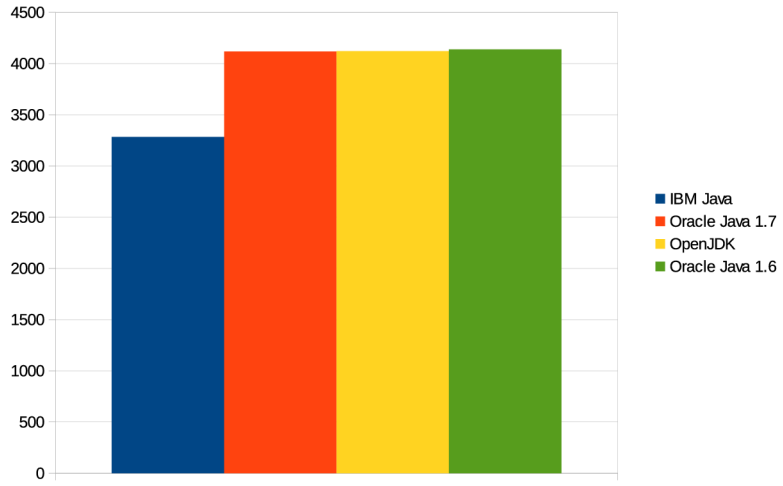
Nejlépe se z testovaných nástrojů umístil stroj od IBM, ostatní stroje poté se skoro totožným výsledkem sdílí druhé až třetí místo.

Dobrý výsledek pro IBM je hlavně dán výrazně lepším provedením logických operací a aritmetických operací sčítání a odčítání u datového typu BigDecimal. Dobře si také poradil s násobením typů BigInteger a BigDecimal při více bitových argumentech.

Ve zkoumaných aritmetických operacích můžeme také vidět zlepšení novější verze Oracle oproti předcházející. Naopak logické operace ukazují zhoršení. Ve výsledku je výkon obou verzí srovnatelný. Trochu ve prospěch novější verze, které trvá v průměru operace o 27 ns méně.

V předešlém porovnání strojů OpenJDK a Oracle 1.7 bylo také zjištěno, že uvedené stroje jsou výkonově velmi blízko, což je vidět i na znázorněném grafu, stroje se liší o zanedbatelné 3 ns.

Průměrný čas pro vykonání operace pak činí 3283 ns pro IBM, 4117 ns pro Oracle 1.7, 4120 ns pro OpenJDK a 4138 ns pro Oracle 1.6.



Obrázek 5.20: Kompozitní výsledek předcházejících testů

Kapitola 6

Závěr

Podařilo se implementovat nástroj, který umožňuje měření výkonnosti jednotlivých java virtuálních strojů při použití tříd z balíčku java.math.

V práci byly srovnány výsledky měření, které byly získány použitím tohoto nástroje na různých implementacích java virtuálních strojů. Naměřené výsledky jsou zejména vhodné pro správce aplikací, jejichž hlavní náplní jsou výpočty s čísly s libovolnou přesností. Výsledky uvedené v této práci jim mohou pomoci s výběrem implementace java virtuálního stroje pro běh aplikace. A správci mohou ušetřit peníze, které by museli vynaložit pro nákup nového technického vybavení, aby získaly větší výpočetní výkon.

K vylepšení nástroje by mohlo přispět rozšíření testů na další metody z tříd BigInteger a BigDecimal. Jako vhodné pokračování také vidím vytvoření nástroje, který by sám konfiguroval a spouštěl jednotlivé testy. Nástroj by také mohl automatizovaně vyhodnocovat výsledky měření, případně by převáděl výsledky do grafické podoby.

Literatura

[1] Bloch, J. *Effective Java*. 2. vyd. Boston: Pearson Education, 2008. ISBN 978-0-321-35668-0.

[2] Boyer, B. *Robust Java benchmarking, Part 1: Issues* [online]. 2008 [cit. 2014-5-3]. Dostupné z: http://www.ibm.com/developerworks/java/library/j-benchmark1/j-benchmark1-pdf.pdf.

[3] Dominik Brodowski. *Cpufreq-set(1) — Linux man page* [online]. [cit. 2014-5-10]. Dostupné z: http://linux.die.net/man/1/cpufreq-set.

[4] Edinburgh Parallel Computing Centre. *The Java Grande Forum Benchmark Suite* [online]. Last revised on Feb 5, 2002 [cit. 2014-4-8]. Dostupné z: http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html.

[5] IBM. *IBM developer kits* [online]. [cit. 2014-5-11]. Dostupné z: http://www.ibm.com/developerworks/java/jdk/.

[6] National Institute of Standards and Technology. *Scimark 2.0* [online]. Last revised on Mar 31, 2004 [cit. 2014-4-4]. Dostupné z: http://math.nist.gov/scimark2/.

[7] Oaks, S. *Java Performance: The Definitive Guide*. 1. vyd. Sebastopol: O'Reilly Media, 2014. ISBN 978-1-449-35845-7.

[8] OpenJDK. *Code Tools: jmh* [online]. Last revised on Apr 16, 2014 [cit. 2014-4-25]. Dostupné z: http://openjdk.java.net/projects/code-tools/jmh/.

[9] OpenJDK. *JDK 7* [online]. [cit. 2014-5-11]. Dostupné z: http://openjdk.java.net/projects/jdk7/.

[10] Oracle. *Class BigDecimal* [online]. Last revised on Mar 18, 2014 [cit. 2014-4-14]. Dostupné z: http://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html.

[11] Oracle. *Class BigInteger* [online]. Last revised on Mar 18, 2014 [cit. 2014-4-14]. Dostupné z: http://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html.

[12] Oracle. *Java SE 6 Downloads* [online]. [cit. 2014-5-11]. Dostupné z: http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase6-419409.html.

[13] Oracle. *Java SE 7 Downloads* [online]. [cit. 2014-5-11]. Dostupné z: http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase7-521261.html.

[14] Oracle. *Package java.math* [online]. Last revised on Mar 18, 2014 [cit. 2014-4-12]. Dostupné z: http://docs.oracle.com/javase/8/docs/api/java/math/package-summary.html.

[15] System Performance Evaluation Cooperative. *SPECjvm2008* [online]. Last revised on Aug 22, 2013 [cit. 2014-4-6]. Dostupné z: http://www.spec.org/jvm2008/.

[16] Wikipedia. *Arbitrary-precision arithmetic* [online]. Last revised on Mar 26, 2014 [cit. 2014-4-12]. Dostupné z: http://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic.

Příloha A

Obsah CD

Obsahem přiloženého CD jsou adresáře a soubory s následujícím obsahem.

- **bin** – obsahuje spustitelný soubor vytvořeného nástroje
- **src** – zdrojové kódy vytvořeného nástroje
- **results** – výsledky ve formátu csv, které jsou diskutovány v kapitole 5
- **tex** – soubory se zdrojovým kódem pro LaTeXobsahující text této práce
- **readme.txt** – soubor obsahující jednoduché instrukce použití nástroje